**Rechnerarchitektur Serie 5** Patrick Stöckli  
 16-942-468

Hoeun Yu

***Aufgabe 1: Statische Sprungvorhersage***

Mit «Predict not taken» gehen wir davon aus, dass bei einem Branch nicht gesprungen wird. Beim gegebenen Code wird aber insgesamt 1024 beim Branch gesprungen, weswegen die Vorhersage dann falsch ist. Erst beim letzten Durchlauf, bei welchem die Schlaufe beendet wird, wird richtig vorhergesagt. Dies hängt damit zusammen, dass «Predict not taken» nicht gut mit Branch-Entscheidungen am Ende einer Schlaufe funktionieren.

Eine optimierte Version sieht beispielsweise folgendermassen aus:

addi $s1, $zero, 1024

loop:  
addi $s1, $s1, -1  
jal subroutine  
beq $s1, $zero, exit  
j loop

exit:

//nächste Funktion oder Codeteile, irrelevant für Aufgabe

Der Vorteil bei dieser Version ist, dass die Vorhersage bei den 1024 Loopdurchgängen eintrifft und erst beim letzten Durchlauf, bei welchem der Loop abgebrochen wird, nicht eintrifft. Hier findet die Branch-Entscheidung am Anfang des Loops statt, was sich besser mit dem «Predict not taken»-Ansatz verträgt.

***Aufgabe 2: Antidependencies***

Bei einer Antidependenz ist eine spätere Instruktion früher fertig als eine frühere (z.B. weil die frühere Operation eine Multiplikation durchführt, welche mehrere Zyklen dauert) und dadurch einen Input-Wert in einem Register überschreibt, welcher die frühere Instruktion als Input verwendet. Dadurch wird natürlich ein falscher Input verwendet. Es handelt sich also um eine Prinzipienumkehr: Statt wie bei normalen Abhängigkeiten das Problem von read before write zu haben, haben wir hier ein write before read-Problem.

Ein Beispiel:

R1 = R1 \* R2  
R1 = R1 + 24

Hier ist eine Antidependenz vorhanden. Die Addition ist schneller fertig als die Multiplikation, weshalb für die Multiplikation bereits der addierte Wert in R1 verwendet wird. Dadurch ist natürlich die Multiplikation verwendet, da quasi folgende Operation ausgeführt wird: R1 = (R1 + 24) \* R2.

***Aufgabe 3: Branch Delay Slots***

Im Falle einer Branch-Instruktion kann normalerweise die nächste Instruktion nicht direkt bestimmt werden. Dies, da die Bedingungen für einen möglichen Sprung zuerst ausgewertet werden müssen. Bis diese Auswertung verfügbar ist, muss eine Wartezeit überbrückt werden. Diese Wartezeit wird Branch Delay Slot genannt. Dieser Slot kann entweder mittels Stall überbrückt werden, oder aber können Instruktionen durchgeführt werden, welche keine Abhängigkeiten haben.

***Aufgabe 4: Dynamische Sprungvorhersage***

Der gegebene C-Code lässt sich mit folgender MIPS-Programmierung ausdrücken:

addi $s1, $zero, 0 //setze i = 0

outerFor: //Beginn der i-For-Schleife

addi $s2, $zero, 0 //setze j = 0  
innerFor: //Beginn der j-For-Schleife

//Funktion für t = t + i \* j, da für Aufgabe irrelevant

addi $s2, $s2, 1 //j++

bne $s2, 8, innerFor //Solange j > 8 wird Loop durchgeführt

addi $s1, $s1, 1 //i++  
bne $s1, 8, outerFor //Solange i > 8 wird Loop durchgeführt

***Aufgabe 5: Multiple-Issue Prozessoren***

* Pipelining: Diese Aussage ist falsch. Pipelining erlaubt die Ausführung von mehreren Instruktionen zur selben Zeit. Grund dafür ist der Fakt, dass die meisten Instruktionen nicht in einem Zyklus ausgeführt werden, sondern mehrere Schritte benötigen. Diese Schritte können aufgesplittet werden und solange durch das Pipelining pro Zyklus für unterschiedliche Instruktionen unterschiedliche Schritte ausgeführt werden, gibt es keine Probleme.
* Multiple-issue processors: Diese Aussage ist richtig. Das klassische Pipelining wird durch das hinzufügen von fundamentalen Elementen erweitert. Beispielsweise führt eine zusätzliche ALU dazu, dass in einem Zyklus zwei Instruktionen ihren EX-Schritt durchführen können. Diese neuen Elemente brauchen aber auch zusätzliche Steuerungselemente, um beispielsweise strukturelle Konflikte zu vermeiden. Ausserdem gibt es zwei Typen: statische und dynamische Multiple-issue Prozessoren.
* SIMD: Diese Aussage ist richtig. Der Daten-Parallelismus wird zum Zeitpunkt des Kompilierens bestimmt.